Des signaux aux symphonies : pour une modélisation homogène des objets sonores

David Janin, Myriam Desainte-Catherine

DES SIGNAUX AUX SYMPHONIES: POUR UNE MODÉLISATION HOMOGÈNE DES OBJETS SONORES

David Janin, Myriam DeSainte-Catherine*

LaBRI, CNRS UMR 5800
INRIA Bordeaux Sud-Ouest
Bordeaux INP
Université de Bordeaux,
F-33405 Talence
janin|myriam@labri.fr

Résumé

Peut-on synchroniser, mixer, combiner des objets sonores de natures diverses tels que des sons, des notes, des accords, des mélodies, tout en faisant abstraction de leur nature ? Il faut pour cela un modèle qui permet de décrire de façon uniforme les agencements temporels qu'on peut souhaiter obtenir en combinant ces objets. Du signal à la symphonie, les tuiles media temporisées polymorphes permettent cela.

1. INTRODUCTION

Problématique.

Dans le domaine de l'informatique musicale, on dispose aujourd'hui d'une variété d'outils tout à fait efficaces. Ensemble, ils couvrent un large spectre des usages possibles de l'informatique appliquée à la musique et au son, de l'analyse à la synthèse musicale, en passant par la capture, l'ordonnancement ou la transformation temps réel de signaux audio ou de gestes musicaux pour les approches programmatiques). Néanmoins, les structures de données manipulées, tout aussi variées, rendent l'interopérabilité de tous ces outils particulièrement délicate à assurer.

Ces outils manipulent-ils des signaux, des notes, des évènements ? Sur quelle échelle de temps ? Quelles horloges ? Les critères et les paramètres définissant la nature et les caractéristiques de ces objets media sont nombreux. Chaque pont logiciel doit faire l'objet d'un développement spécifique, souvent coûteux. Les objets manipulés doivent être traduits d'un format de représentation dans un autre. Lorsque ces formats ne sont pas compatibles, ces traductions ne peuvent être que partielles, perdant ainsi de l'information.

* soutenu par le projet ANR-12-CORD-0009

Plus ennuyeux, quand bien même de tels ponts sont réalisés, le concepteur artistique de ces systèmes musicaux, qui souhaite faire un usage combiné de ces outils, peut voir sa créativité bridée par la nécessaire compréhension de nombreux détails techniques, qui, orthogonaux au processus créatifs, peuvent se révéler, a posteriori, parfaitement inutiles pour la création elle-même.

Idéalement, l'agencement d'objets sonores unitaires, qui peuvent être, selon le niveau d'abstraction choisi, des sons, des notes, des motifs mélodiques, des accords, des agglomérats, devrait pouvoir être conçu et décrit indépendamment de la structure de ses objets. Les objets plus complexes qui résulteraient de la combinaison de ces objets unitaires, changeant ainsi de niveau d'abstraction, devraient pouvoir être traités de même. Ses objets sonores sont-ils caractérisés par des hauteurs ? Des textures ? Des durées ? Des paramètres continus ? Discrets ? Sont-ils codés en évènements MIDI ou OSC ? Sont-ils liés à des fichiers sonores ? Des fonctions de synthèse ? Sont-ils homogènes ? Qu'importe, un large éventail d'opérations de combinaisons devrait déjà permettre de les manipuler uniformément, de façon simple et homogène.

Aujourd'hui, force est de constater que de tels formalismes n'existent pas encore. La manipulation combinée de flux de natures et de fréquences différentes reste largement une affaire de techniciens experts ou de compositeurs disposant d'une solide connaissance de l'outil informatique. Pourtant, l'accès à ces outils peut encore être facilité.

Un modèle uniforme d'agencement temporel.

Dans un travail de recherche amont, nous cherchons ici à concevoir un modèle programmatique, c'est à dire un modèle qui pourra être utilisé dans les langages de programmation, qui permet une description uniforme des objets sonores : des signaux aux symphonies pour

reprendre une image de Paul Hudak [10]. A terme, ce modèle doit permettre le développement d'interfaces plus homogènes, transversales aux nombreux outils disponibles, pour la représentation, la combinaison et le traitement de ces objets sonores. En s'affranchissant largement de la nature des objets sonores, cette interface doit permettre ainsi de simplifier leur manipulation. Les outils résultant seront alors plus accessibles aux compositeurs eux-mêmes.

Pour faire cela, dans une approche formelle, mathématiquement bien fondée, nous cherchons à développer une sorte de schema d'usage abstrait qui permettrait de voir et de manipuler de façon uniforme les signaux, les notes, les mélodies, les mouvements, etc. En intégrant la modélisation multi-échelle, les spécialisations nécessaires à chacun des niveaux d'instanciation possibles, viendraient *a posteriori*, et seulement lorsque c'est absolument indispensable, raffiner ce modèle. Autrement dit, en faisant une analogie avec la modélisation orientée objet, nous nous proposons donc de définir une sorte de classe abstraite permettant de décrire la structuration et les méthodes communes à l'ensemble de ces objets sonores : leur synchronisation, c'est à dire, leur placement temporel relatif les uns aux autres.

Ces travaux se placent à l'intersection des approches structuralistes telles que la théorie générative de la musique tonale [17], mathématiquement bien fondée, et des travaux plus pragmatiques autour de la programmation de scenarii interactifs par contraintes de placement temporel tel que, par exemple, la plateforme *i-score* [2, 23].

Le modèle des tuiles musicales.

En théorie, le modèle des tuiles avec superpositions partielles [15, 5] doit pouvoir remplir ce rôle.

D'un point de vue abstrait, ce modèle permet en effet de faire une description hors temps des pièces musicales. Dans un processus de composition musicale, les œuvres peuvent en effet y être décrites à l'aide d'aller-retours dans le temps musical. L'exécution d'une pièce ainsi modélisée, qui necessite de ré-ordonner les objets sonores dans le temps musical, est alors laissée à la charge de l'ordinateur (voir [3, 14]).

Une expérimentation [4], réalisée en C++, qui s'appuie sur la *libAudioStream* [19], permet déjà de manipuler des *flux audio tuilés*. Une autre, réalisée en Haskell au dessus du DSL *Euterpea* [10], permet aussi d'expérimenter le tuilage sur des suites d'évènements MIDI [12]. Un re-codage de cette dernière [3] démontre même que la modélisation par tuilage est tout aussi primitive que la modélisation, plus classique, par produits séquentiels et parallèles [9]. D'autres codages expérimentaux sont par ailleurs à l'étude en CLOS pour OpenMusic[6, 1], et, tout récemment, en OCaml.

En pratique, ces implémentations ne sauraient être que des prototypes tant que la pertinence de notre approche n'aura pas été démontrée avec force exemples et expérimentations. L'objet de cet article est donc de proposer une telle expérimentation.

Plus précisément, nous montrons que le modèle des tuiles, appliqué à la représentation des signaux, permet de coder, de façon simple et élégante, un traitement jusqu'alors considéré comme relevant de l'audio numérique [22] : la dilatation et contraction de signaux audio par des techniques de type synthèse granulaire. Surprise, au niveau symbolique, cette même transformation nous permettait déjà de générer toute une variété de rythmes à partir d'une marche binaire, de la valse au *tumbao* de la salsa.

Remarque : prototyper en Haskell.

Tous les exemples et les définitions qui suivent sont écrits dans le langage de programmation fonctionnel Haskell [11]. Remarquons que l'utilisation des langages de programmation fonctionnelle pour écrire et décrire du son et de la musique n'est pas nouvelle comme l'illustre les langages Fugue [7], Elody [18], OpenMusic [6] ou Faust [20].

Déclaratifs, les langages fonctionnels visent plus à une description structurelle de la musique. En un sens, ils traitent bien plus de la composition musicale ; une activité qui procède largement d'une description hors-temps [14]. Leur usage contraste avec les langages procéduraux tels que les langages C ou C++ qui sont plus utilisés pour une description de la production musicale, c'est à dire la performance musicale ; une activité dans le temps.

Dans cet article, l'utilisation des classes de types d'Haskell nous permet, en quelques lignes, de rappeler la définition des signaux et de définir simplement leur plongement dans les tuiles. Nous pouvons ensuite donner une description uniforme, c'est-à-dire applicable à d'autres types d'objets sonores, de manipulation jusqu'alors réservés au traitement du signal.

2. SIGNAUX

Dans cette première partie, nous passons en revue la notion de signal telle qu'elle peut être manipulée en audio-numérique, c'est à dire la notion de signal numérisé. Nous montrerons dans les parties suivantes quel bénéfice il y a à voir les signaux comme des cas particuliers de tuiles.

Propriétés élémentaires.

La déclaration de la classe *Signal a* ci-dessous décrit dans quelle condition un type de données *a* peut être manipulé comme un type de signal.

```
class Signal a where
  type SigVal a :: *
```

type $SigDur\ a :: *$
$empty :: SigDur\ a \to a$
$const :: SigVal\ a \to SigDur\ a \to a$
$(:+:) :: a \to a \to a$
$eval :: a \to SigDur\ a \to Maybe\ (SigVal\ a)$

Le type $SigVal\ a$ est le type de valeurs instantanées possibles pour les signaux de type a. Le type $SigDur\ a$ est le type de durées possibles pour ces signaux. Les fonctions *empty* et *const* sont les constructeurs de bases de signaux de type a permettant de définir les signaux vides et les signaux constants. Ces signaux de base peuvent ensuite être composés séquentiellement avec l'opérateur infixe $(:+:)$. La fonction *eval* permet quant à elle de lire la valeur d'un signal à un instant donné.

La durée $dur\ s$ d'un signal s est définie comme la durée entre ce « début » et cette « fin ». Avec le type $dur :: (Signal\ a) \Rightarrow a \to SigDur\ a$, sous l'hypothèse, naturelle, que les durées peuvent être additionnées, la fonction *dur* satisfait les equations suivantes :

$$dur\ (empty\ d) \quad == \quad d \tag{1}$$
$$dur\ (const\ a\ d) \quad == \quad d \tag{2}$$
$$dur\ (s_1 :+: s_2) \quad == \quad (dur\ s_1) + (dur\ s_2) \tag{3}$$

valides pour toute valeur a, toute durée d et tout signaux s_1 et s_2.

Par exemple, avec des durées rationnelles, le signal $s = (const\ a_1\ 2) :+: (empty\ 1) :+: (const\ a_2\ 3)$ est décrit Figure 1. Dans cette figure, une barre verticale située

Figure 1. Un exemple de signal s

au dessus du signal marque son « début » et une barre verticale située au dessous du signal marque sa « fin ».

Evaluation et égalité de valeurs.

La fonction d'évaluation, détaillée ci-dessous, induit une équivalence sémantique sur les signaux. En effet, en supposant l'égalité de valeur définie sur les types $SigVal\ a$ et $SigDur\ a$, on dit que deux signaux s_1 et s_2 de type a ont même valeur, ce qui est noté $s_1 == s_2$, lorsque $dur\ s_1 == dur\ s_2$ et $eval\ s_1\ d == eval\ s_2\ d$ pour toute durée d.

Intuitivement, chaque signal peut être vu comme une fonction du temps, constante par morceaux, c'est-à-dire une suite d'échantillons de longueur possiblement variable. L'instant « initial » de la fonction *eval*, de valeur 0, correspond à la date du« début » du signal. Deux signaux ont donc même valeur lorsqu'ils ont même durée et qu'ils codent la même fonction *eval*.

La date passée en paramètre de la fonction *eval* est bien décrite par une valeur de type $SigDur\ a$ qui mesure la distance, en durée, de cette date par rapport à l'instant « initial ». Le type du résultat de la fonction *eval* est bien le type $Maybe\ (SigVal\ a)$ afin de rendre compte de la valeur « vide » qu'elle peut renvoyer, par exemple sur un signal vide $empty\ d$. En effet, par definition du constructeur de type *Maybe* en Haskell, les valeurs possibles de type $Maybe\ b$ sont soit la valeur *Nothing* codant la valeur « vide », soit une valeur de la forme $Just\ v$ codant la valeur « non vide » v de type b.

Plus précisément, la sémantique de la fonction *eval* est définie par les règles suivantes. Pour toute valeur de x, on a :

- $eval\ (empty\ d)\ x$ qui vaut *Nothing*,
- $eval\ (const\ a\ d)\ x$ qui vaut $Just\ a$ lorsque $0 \leqslant x < d$, et qui vaut *Nothing* lorsque $x < 0$ ou $d \leqslant x$,
- et, pour tout signal s_1 et s_2, $eval\ (s_1 :+: s_2)\ x$ qui vaut $eval\ s_1\ x$ lorsque $x < dur\ s_1$ et qui vaut $eval\ s_2\ (x - (dur\ s_1))$ lorsque $dur\ s_1 \leqslant x$.

En particulier, la valeur instantanée $eval\ s\ x$ d'un signal s à l'instant x vaut *Nothing* dès lors que $x < 0$ ou $dur\ s \leqslant x$.

Par exemple, la fonction d'évaluation appliquée au signal s de l'exemple de la Figure 1 est illustrée dans la Figure 2.

x	-1	0	1	2	3	4	5	6
$eval\ s\ x$	*Nothing*	$Just\ a_1$	$Just\ a_1$	*Nothing*	$Just\ a_2$	$Just\ a_2$	$Just\ a_2$	*Nothing*

Figure 2. Valeurs instantanées du signal s

A partir de ces règles, on vérifie facilement que l'opération de composition séquentielle est, modulo égalité de valeur, associative.

Sous l'hypothèse, naturelle, que le type des durées admet un zéro, noté 0, on vérifie aussi que le signal $empty\ 0$ est équivalent au signal $const\ v\ 0$ pour n'importe quelle valeur v. De plus, ce signal vide est l'élément neutre pour le produit séquentiel.

Autrement dit, d'un point de vue sémantique, sous des hypothèses naturelles, l'ensemble des signaux d'un type donné muni du produit de composition séquentielle forme un monoïde.

Produits externes, produits internes.

Plusieurs notions de produits de signaux peuvent être définies à partir des lignes qui précèdent. Nous donnons ici les spécifications de deux des produits les plus emblématiques : le produit parallèle qui réalise une

sorte de jointure libre de deux signaux quelconques, le produit de fusion qui réalise la fusion parallèle de deux signaux de même type.

Produit parallèle.

Pour tout type de signaux a et b, on peut aussi définir le type de signal $ParSig\ a\ b$ tel que $Sigval\ (ParSig\ a\ b)$ est égal à $(Maybe\ (SigVal\ a), Maybe\ (SigVal\ b))$ c'est à dire, avec la notation Haskell, au produit cartésien des types $Maybe\ (SigVal\ a)$ et $Maybe\ (SigVal\ b)$, et tel qu'il existe une fonction

$$parSig :: a \to b \to ParSig\ a\ b$$

telle que, pour tout signal $s_1 :: a$ et $s_2 :: b$, on a

$$dur\ (parSig\ s_1\ s_2) = max\ (dur\ s_1)\ (dur\ s_2)$$

avec, pour toute durée d :

$$eval\ (parSig\ s_1\ s_2)\ d = (eval\ s_1\ d, eval\ s_2\ d)$$

Autrement dit, l'opérateur *parSig* apparait comme un opérateur de composition parallèle, le type $ParSig\ a\ b$ représentant le type obtenu par composition parallèle de signaux de type a et de type b.

Un exemple de produit parallèle est décrit Figure 3. Avec la même convention graphique pour marquer les

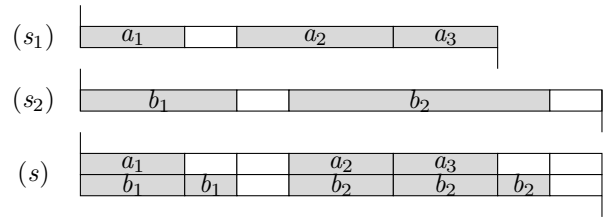

Figure 3. Un produit $s = parSig\ s_1\ s_2$.

« débuts » et « fins » des signaux, on constate que la durée du produit est bien égale à la durée de l'argument le plus long.

Produit de fusion.

Le produit évoqué ci-dessus est un produit externe au sens où il se définit via des combinaisons de types de signaux possiblement distincts.

Le produit de fusion présenté ici est, au contraire, un produit interne à tout type de signaux, dès lors que qu'une fonction primitive de fusion des valeurs instantanées est donnée en paramètre.

Plus précisément, en supposant que le type de signal a est équipé d'une fonction

```
mergeVal : : SigVal a -> SigVal a -> SigVal a
```

on peut définir la fonction

$$mergeSig :: (Signal\ a) \Rightarrow a \to a \to a$$

de telle sorte que pour tout signal s_1 et s_2 de même type, on a

$$dur\ (mergeSig\ s_1\ s_2) = max\ (dur\ s_1)\ (dur\ s_2)$$

avec, pour toute durée d :

- ▷ si $eval\ s_1\ d$ vaut $Nothing$ alors on doit avoir $eval\ (mergeSig\ s_1\ s_2)\ d$ qui vaut $eval\ s_2\ d$,
- ▷ si $eval\ s_2\ d$ vaut $Nothing$ alors on doit avoir $eval\ (mergeSig\ s_1\ s_2)\ d$ qui vaut $eval\ s_1\ d$,
- ▷ si $eval\ s_1\ d$ vaut $Just\ a_1$ et si $eval\ s_2\ d$ vaut $Just\ a_2$ alors $eval\ (mergeSig\ s_1\ s_2)\ d$ vaut $Just\ (mergeVal\ a_1\ a_2)$.

Autrement dit, le produit de fusion défini par *mergeSig* consiste à réduire, via la fonction *mergeVal* qui apparaît comme le paramètre de cette opération de fusion, deux signaux « lus » en parallèle.

Un exemple de produit de fusion est décrit Figure 4, en notant $a_1 + a_2$ la valeur de $mergeVal\ a_1\ a_2$ pour toute valeur a_1 et a_2 de type $SigVal\ a$. On constate que

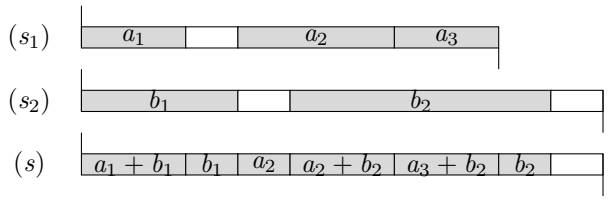

Figure 4. Une fusion $s = mergeSig\ s_1\ s_2$.

la durée du produit de fusion est bien égale à la durée de l'argument le plus long.

Remarque. Observons que la valeur de signal « vide », codée par *Nothing*, a fonction de neutre lors de la fusion de deux signaux. Plus précisément, on peut vérifier que $s == mergeSig\ s\ (empty\ 0) == mergeSig\ (empty\ 0)\ s$ pour tout signal s.

Ainsi, sous l'hypothèse, naturelle, que la fonction *mergeVal* soit associative et commutative, on peut aussi constater que l'ensemble des signaux équipé du produit de fusion forme un monoïde commutatif.

Limite de la modélisation par signaux.

Les opérations définies ci-dessus peuvent déjà être utilisées pour la composition musicale. Elle apparaissent peu ou prou dans la quasi totalité des langages de manipulation de signaux ou de séquences musicales (voir par exemple [10] ou [21]).

Les signaux ou séquences finis peuvent en effet être combinés en séquence ou en parallèle, l'insertion de signaux vides (ou silences) permettant de caler de façon adéquate les objets manipulés. Autrement dit, la

construction de pièce musicale complexe à partir des opérations définies ci-dessus est parfaitement possible.

Cependant, comme déjà observé dans [15, 16, 13], la réutilisabilité des programmes écrits avec cette approche est discutable. Par exemple, le mixage, c'est à dire le placement temporel puis la fusion de signaux audio ou de voix musicales, peut s'avérer non modulaire lorsqu'il doit être codé par l'insertion de silences comme l'illustre, en particulier, le codage de la notion d'anacrouse.

En effet, le démarrage d'une voix musicale en anacrouse décrit, en musique, un départ de cette voix en anticipation du premier temps d'une ligne musicale. Néanmoins, d'un point de vue plus abstrait, toutes les voix mises en œuvre s'accordent sur ce même premier temps : il s'agit donc aussi d'un départ simultanée.

Dans un codage de type signal, le changement de la durée de cette anacrouse risque de décaler les voix codées de façon inappropriées. Une implémentation, générique et réutilisable de cette notion abstraite de départs simultanés est bien sur possible. Il suffit de mettre la durée des anacrouses en paramètre. Mais cette solution relève d'un codage adhoc qui peut se révéler laborieux et plus difficile à maintenir [15, 16].

Le tuilage des signaux, rappelé ci-dessous, intègre dans le modèle toute la problématique du positionnement temporel des signaux les uns par rapport aux autres. Il se révèle donc bien plus modulaire.

3. SIGNAUX TUILÉS

La notion de signal tuilé apparaît à mi chemin entre l'approche formelle proposée par Hudak[9] et l'approche plus intuitive de Desain et Honing autour d'une adaptation du langage Logo à la musique [8]. Récemment formalisé [5], deux implémentations en ont déjà été proposées [13, 12, 3]. Nous rappelons ci-dessous la définition de ces tuilages avec superpositions partielles et donnons quelques exemples simples.

Propriétés élémentaires.

Un signal tuilé peut être simplement vu comme un signal enrichi de deux marqueurs de synchronization *Pre* et *Post* (voir Figure 5). En un sens, le marqueur

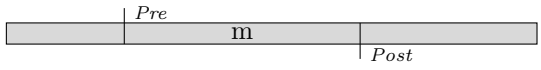

Figure 5. Un signal tuilé type.

Pre code une notion abstraite de « début », le marqueur *Post* code une notion abstraite de « fin », deux notions liées à l'usage qu'on va faire du signal.

On s'affranchit ainsi des notions plus concrètes de « début » ou de « fin » de signaux vues jusqu'alors, deux notions bien plus liées à la nature des signaux manipulés. En particulier, un« début » abstrait peut même être postérieur à une fin abstraite (voir Figure 6).

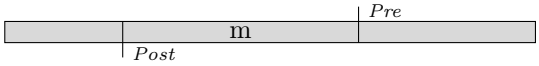

Figure 6. Un autre signal tuilé type.

Comme nous le verrons dans la suite avec la notion de produit tuilé, le marqueur *Pre* indique comment vont se placer les objets sonores situés « avant », le marqueur *Post* indique comment vont se placer les objets sonores situés « après ».

Bien entendu, un même signal peut avoir plusieurs « tuilages » possibles comme illustré Figure 7 ci-dessous.

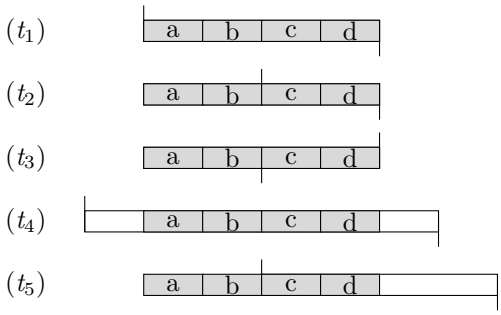

Figure 7. Plusieurs « tuilage » pour un même signal.

La fonction d'évaluation *eval* reste définie de la même façon, l'instant 0 de référence devenant le point de synchronisation *Pre*. On illustre cette différence Figure 8 qui donne quelques valeurs instantanées de la tuile t_5 définie ci-dessus.

x	-4	-3	-2	-1	0	1	2	3
eval t_5 x	*Nothing*	*Nothing*	*Just a*	*Just b*	*Just c*	*Just d*	*Nothing*	*Nothing*

Figure 8. Valeurs instantanées de la tuile t_5

Dans la suite, en notant *Tile a* les signaux tuilés dont le type *SigVal* de valeur de signal vaut *a*, on suppose qu'on dispose comme avec les signaux de la fonction de durée

$$dur :: Tile\ a \rightarrow Rational$$

qui donne, pour chaque tuile, la distance entre les marqueurs *Pre* et *Post*. Remarquons que pour cette fonction *dur*, les équations (1)-(2) de la Section 2 restent valides. C'est-à-dire

$$dur\ (empty\ d) == d \text{ et } dur\ (const\ v\ d) == d.$$

Dans ce cas, on autorise cependant d à être négatif. Comme nous allons le voir dans un instant avec la notion de produit tuilé, on peut aussi générer tout tuilage par produits de signaux non tuilés (ou trivialement tuilés) et signaux vides positifs ou négatifs.

Plus précisément, en interprétant simplement le produit d'un signal par un délai positif ou négatif comme un déplacement, arrière ou avant, des marqueurs de début et de fin, tous les signaux de la Figure 7 peuvent être obtenus à partir du premier de la façon suivante :

$t_2 = \mathit{empty}\ (-2)\ \%\ t_1$
$t_3 = \mathit{empty}\ (-4)\ \%\ t_1\ \%\ \mathit{empty}\ (-2)$
$t_4 = \mathit{empty}\ 1\ \%\ t_1\ \%\ \mathit{empty}\ 1$
$t_5 = \mathit{empty}\ (-2)\ \%\ t_1\ \%\ \mathit{empty}\ 2$

en notant toujours % le produit tuilé qui, comme nous allons le voir, généralise tout à la fois le produit séquentiel et le produit de fusion.

Produits tuilés.

En toute généralité, étant fixé une fonction paramètre *mergeVal* de fusion des valeurs de signaux, le produit de deux signaux tuilés de type *Tile a* se définit en deux étapes :

(1) synchronization du marqueur de sortie du premier signal avec le marqueur d'entrée du second signal,

(2) fusion des signaux ainsi positionnés,

le marqueur d'entrée résultant étant celui du premier signal tuilé, le marqueur de sortie résultant étant celui du second signal tuile.

Cette opération de produit est décrite Figure 9 ci-dessous.

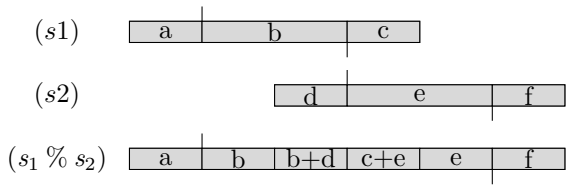

Figure 9. Le produit tuilé de deux signaux tuilés.

Remarque. Lorsque les signaux sont trivialement tuilés, c'est à dire lorsque *Pre* marque le début des signaux et *Post* la fin des signaux, alors le produit tuilé coincide avec le produit séquentiel. Cette remarque justifie qu'on puisse utiliser, pour la composition séquentielle de signaux comme pour le produit tuilé de signaux tuilés le même opérateur : %. Plus généralement, on suppose qu'on a bien $\mathit{Signal}\ (\mathit{Tile}\ a)$, c'est à dire le type *Tile a* satisfait la spécification des types signaux avec l'égalité $\mathit{SigVal}\ (\mathit{Tile}\ a) = a$. On dispose donc des signaux vides *empty d* ou les signaux constants *const a d* de durée d trivialement tuilés. Par la suite, dans tous nos exemples on suppose que $\mathit{SigDur}\ (\mathit{Tile}\ a) = \mathit{Rational}$, c'est à dire que les durées sont spécifiées dans le type *Rational*.

D'autres exemples de produits tuilés construits à partir des signaux tuilés de la Figure 7 sont décrits Figure 10 ci-dessous.

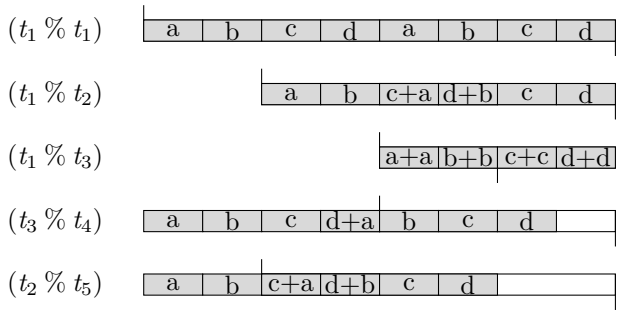

Figure 10. Exemples de produits tuilés construits à partir des signaux tuilés de la Figure 7.

Reset, co-reset, inverse et resync.

Les exemples de tuilages ci-dessus invitent à la définition de trois opérateurs particuliers sur les signaux tuilés : le reset, noté *re*, le co-reset, note *co*, et l'inverse, noté *inv*, definis par :

$\mathit{re}\ s = s\ \%\ \mathit{empty}\ (-(\mathit{dur}\ s))$
$\mathit{co}\ s = \mathit{empty}\ (-(\mathit{dur}\ s))\ \%\ s$
$\mathit{inv}\ s = \mathit{empty}\ (-(\mathit{dur}\ s))\ \%\ s\ \%\ \mathit{empty}\ (-(\mathit{dur}\ s))$

Les trois opérations de resynchronisation élémentaires, *re*, *co* et *inv* sont illustrées Figure 11. On vérifie facile-

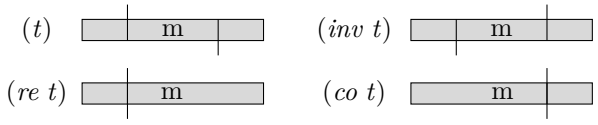

Figure 11. Resynchronisation élémentaires.

ment que les équations sont satisfaites :

$$\begin{aligned} \mathit{inv}\ (\mathit{empty}\ d) &== \mathit{empty}\ (-d) \\ \mathit{inv}\ (\mathit{const}\ v\ d) &== \mathit{const}\ v\ (-d) \\ \mathit{inv}\ (t_1\ \%\ t_2) &== \mathit{inv}\ t_2\ \%\ \mathit{inv}\ t_1 \end{aligned}$$

Nous avons déjà observé que le produit tuilé généralise le produit séquentiel (voir Section 3).

Sous l'hypothèse que la fonction de fusion point à point *mergeVal* soit commutative, le produit de fusion *mergeSig* peut lui aussi être généralisé sur les signaux tuilés en posant :

$\mathit{forkMerge}\ t_1\ t_2 = \mathbf{if}\ (\mathit{dur}\ t_1 < \mathit{dur}\ t_2)\ \mathbf{then}\ (\mathit{re}\ t_1)\ \%\ t_2$
$\mathbf{else}\ (\mathit{re}\ t_2)\ \%\ t_1$

Plus intéressant encore, la notion de signal tuilé étant symétrique, on peut aussi coder une fusion avec synchronisation sur le *Post* plutôt que le *Pre*.

En reprenant l'exemple de produit de fusion donné Figure 4, on donne Figure 12 une illustration de la co-fusion telle qu'induite par une fonction *joinMerge* duale de la fonction *forkMerge*.

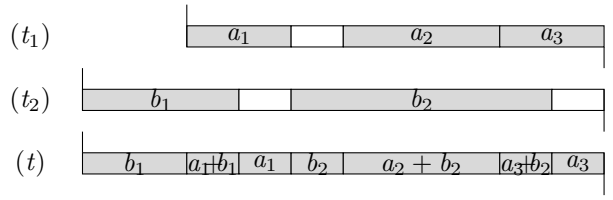

Figure 12. Une co-fusion $t = joinMerge\ t_1\ t_2$.

Le code de la fonction *joinMerge* est donné par :

$$joinMerge\ t_1\ t_2 = \textbf{if}\ (dur\ t_1 > dur\ t_2)\ \textbf{then}\ t_1\ \%\ (co\ t_2)\ \textbf{else}\ t_2\ \%\ (co\ t_1)$$

sous la même hypothèse de commutativité de *mergeVal*.

Plus généralement, on définit la fonction générique de resynchronisation, noté *resync*, en posant :

$$resync\ s\ m\ n = empty\ (-m)\ \%\ s\ \%\ empty\ n$$

La resynchronisation générique est illustrée Figure 13.

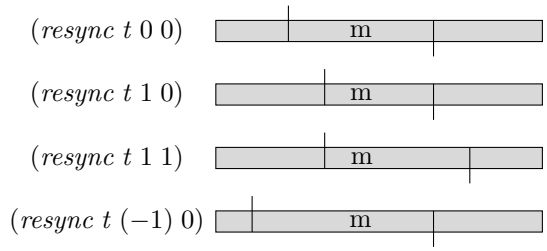

Figure 13. La resynchronisation généralisée par l'exemple.

4. APPLICATIONS

Nous présentons dans cette section quelques utilisations du modèle des flux tuilés pour la manipulation et la transformation de flux temporisé.

Décomposition par valeur.

Les applications présentées dans la suite sont toutes basées sur la procédure de normalisation à la volée présentée dans [3] (voir aussi [14] pour une présentation plus intuitive) qui peut facilement être étendue au signaux tuilés. On en rappelle ici les caractéristiques principales et on en donne une généralisation.

On suppose aussi qu'on dispose d'une fonction

$$first :: Tile\ a \to Maybe\ (Rational)$$

qui renvoie la date, possiblement nulle, du début du flux. Plus précisément, pour toute tuile t on a *first* t défini par :

- ▷ *first* t == *Nothing* lorsque *eval* $t\ x$ == *Nothing* pour toute durée x, c'est à dire, lorsque le signal sous-jacent est vide,
- ▷ *first* t == *Just* d lorsque *eval* $t\ x$ == *Nothing* si $x < d$ et *eval* $t\ d$ == *Just* v.

En particulier :

$$\begin{aligned} first\ (empty\ d) &== Nothing \\ first\ (const\ v\ d) &== \textbf{if}\ (d > 0)\ \textbf{then}\ 0\ \textbf{else}\ d \end{aligned}$$

La procédure de normalisation peut être résumée de la façon suivante. Il existe deux fonctions

$$headT :: Tile\ a \to Tile\ a$$
$$tailT :: Tile\ a \to Tile\ a$$

telles que, pour toute tuile t, on a

$$t == headT\ t\ \%\ tailT\ t$$

c'est à dire que *headT* et *tailT* définissent une décomposition de t. Cette décomposition est illustrée Figure 14. Plus en détail, on suppose que les équations suivantes

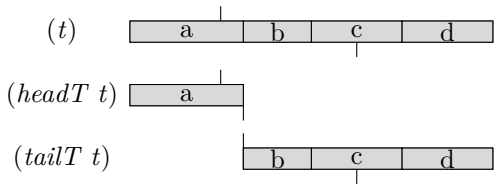

Figure 14. Un exemple de normalisation par *headT* et *tailT*.

sont satisfaites :

- ▷ si *first* t == *Nothing* alors on a *headT* t == t et *tailT* t == *empty* 0,
- ▷ si *first* t == *Just* d_1 alors il existe $d_2 > 0$ et v de type a tels que :
  - ◦ *headT* t == *empty* d_1 % *const* v d_2, c'est à dire que *headT* t contient la première valeur du signal tuilé t,
  - ◦ *first* (*tailT* t) vaut soit *Nothing* soit *Just* 0, c'est à dire que *tailT* t contient l'échantillon suivant, s'il existe,

  avec, dans ce cas, *dur* (*headT* t) maximum, c'est à dire, de façon équivalente, d_2 maximum.

Autrement dit, $headT\ t$ est la tuile de tête correspondant à la première valeur du signal codé par t. En particulier, dans la Figure 14, on suppose implicitement que a et b sont distincts puisque la fin de $headT\ t$ doit correspondre à un changement de valeur.

Remarque. C'est bien une *normalisation à la volée* qui est effectuée par les fonctions $headT$ et $tailT$.

En effet, un signal tuilé peut être défini, en toute généralité, par une succession de produits tuilés de signaux constant et de délais, positifs ou négatifs, c'est-à-dire, une définition en zigzag (voir [14]). Les fonctions $headT$ et $tailT$ permettent alors, de façon incrémentale, de réordonner les valeurs des signaux tuilés en les calculant dans l'ordre de leur apparition temporelle, la première valeur étant codée en tuile dans le résultat de $headT$.

Plus précisément, en Haskell, l'évaluation est paresseuse. La valeur du signal a un instant donné n'est calculée que lorsqu'elle est nécessaire. Pour autant, pour produire la valeur d'un signal à un instant donné, doit-on le calculer à tout instant ? La procédure d'évaluation, qui découle des fonctions $headT$ et $tailT$, consiste justement à n'évaluer le signal, du passé vers le futur, que jusqu'à atteindre l'instant désiré.

En particulier, alors que l'évaluation de $headT$ est complète, l'évaluation de $tailT$ ne fait qu'agglomérer les morceaux de signaux nécessaires à la définition de la suite. On peut ainsi montrer que, sous réserve de certaines hypothèses de calculabilité *first* et *dur*, on aussi traiter le cas de signaux tuilés « infinis « dont la continuation est le fruit d'un calcul qui sera fait plus tard.

Question complexité, on montre dans [3] que cela peut être fait de façon optimale, c'est à dire quasi-linéaire en la taille d'expression des signaux, ce que l'implémentation actuelle confirme.

Décomposition par durée.

Dans les définitions ci-dessus, le point de synchronisation entre $headT$ et $tailT$ est définit par la (fin de la) première valeur d'un signal tuilé. On peut aussi, à la place, fixer ce point de décomposition à partir de sa durée relativement au marqueur de synchronisation d'entrée *Pre*.

Plus précisément, on définit l'extraction de tête par la fonction suivante.

$$
\begin{array}{l}
extHeadT\ t\ d = \textbf{let}\ (h1, t_1) = (headT\ t, tailT\ t) \\
\qquad\qquad\qquad\quad d_1 = dur\ h1 \\
\quad \textbf{in if}\ (d_1 < d)\ \textbf{then}\ h1\ \%\ (extHeadT\ t_1\ (d - d_1)) \\
\qquad \textbf{else case}\ (first\ h1)\ \textbf{of} \\
\qquad\qquad\qquad (Nothing) \rightarrow empty\ d \\
\qquad\qquad\qquad (Just\ v) \quad \rightarrow empty\ d_1 \\
\qquad\qquad\qquad\qquad\qquad\quad \%\ const\ v\ (d - d_1)
\end{array}
$$

La sémantique de l'extraction de tête est illustrée Figure 15.

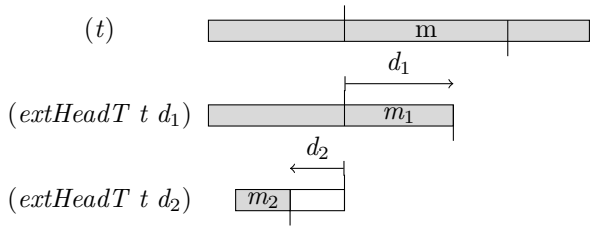

Figure 15. Extraction de tête.

De façon duale, on peut définir l'extraction de queue dont le code est décrit ci-dessous.

$$
\begin{array}{l}
extTailT\ t\ d = \textbf{let}\ (h1, t_1) = (headT\ t, tailT\ t) \\
\qquad\qquad\qquad\quad d_1 = dur\ h1 \\
\quad \textbf{in if}\ (d_1 < d)\ \textbf{then}\ (extTailT\ t_1\ (d - d_1)) \\
\qquad \textbf{else case}\ (first\ h1)\ \textbf{of} \\
\qquad\qquad\qquad (Nothing) \rightarrow empty\ (d_1 - d)\ \%\ t_1 \\
\qquad\qquad\qquad (Just\ v) \quad \rightarrow const\ v\ (d_1 - d)\ \%\ t_1
\end{array}
$$

La sémantique de l'extraction de queue est illustrée Figure 16.

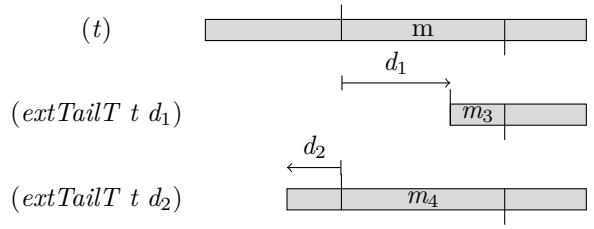

Figure 16. Extraction de queue.

Les fonctions $extHeadT$ et $exTailT$ généralisent bien les fonction de normalisation $headT$ et $tailT$ au sens où, comme on peut s'en convaincre, pour toute tuile t, toute durée d l'équation suivante est satisfaite :

$$t == extHeadT\ t\ d\ \%\ extTailT\ t\ d$$

De plus, on vérifie que le code proposé ci-dessus reste compatible avec le cas de tuile infinie.

Etirement/contraction temporelle.

Nous illustrons maintenant la notion de tuilage par son application à l'étirement/contraction définie par décomposition et re-synthèse « granulaire ».

Plus précisément, nous montrons comment un signal donné m, trivialement tuilé, peut être décomposé puis recomposé à l'aide des fonctions permettant ainsi, comme cela se fait, en synthèse granulaire, de faire varier sa vitesse d'execution sans changer la hauteur des sons [1] .

1 . bien entendu, la technique présentée ici n'est pas parfaite, et produit, en particulier, de nombreux artefacts. . .

La décomposition et la recomposition associée en grains tuilés, peuvent être décrites par :

$$
\begin{array}{l}
\mathit{decompose}\ t\ d\ d_1 = \textbf{case}\ (\mathit{first}\ t)\ \textbf{of} \\
\quad \mathit{Nothing} \quad\quad \rightarrow t \\
\quad \mathit{otherwise} \quad\quad \rightarrow ((\mathit{extHeadT}\ t\ d)\ \%\ (\mathit{delay}\ (d_1 - d))) : \\
\qquad\qquad\qquad\quad (\mathit{grandDec}\ (\mathit{extTailT}\ t\ d_1)\ d_1\ d)
\end{array}
$$

$$
\begin{array}{l}
\mathit{recompose}\ [\,] = \mathit{empty}\ 0 \\
\mathit{recompose}\ (t :: ts) = t\ \%\ ts
\end{array}
$$

La décomposition $\mathit{decompose}\ t\ d\ d_1$ produit, de façon récurrente, une liste de tuiles de longueur totale d, dont la durée n'est que de longueur d_1. La recomposition de ces tuiles produit à nouveau, modulo les superpositions, le signal initial. Autrement dit, pour toute tuile t, toute durée d et d_1, l'équation suivante est satisfaite :

$$\mathit{compose}\ (\mathit{decompose}\ t\ d\ d_1) == t$$

Dans ce cas cependant, il est abusif d'utiliser le symbole d'égalité de valeur ==. L'équivalence ci-dessus n'est vraie que dans la mesure où le signal recomposé est aussi normalisé en un certain sens.

Cette décomposition est illustrée Figure 17.

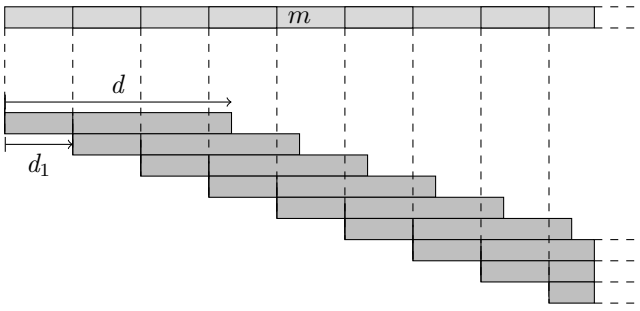

Figure 17. Décomposition et recomposition « granulaire ».

L'effet de re-synthèse granulaire apparait dès lors que la recomposition ne se fait pas avec le même décalage d_1 qui prévaut à la décomposition. Plus précisément, on peut définir la fonction :

$$
\begin{array}{l}
\mathit{stretch}\ t\ d\ d_1\ d_2\ \textbf{case}\ (\mathit{first}\ t)\ \textbf{of} \\
\quad \mathit{Nothing} \quad \rightarrow t \\
\quad \mathit{otherwise} \rightarrow (\mathit{extHeadT}\ t\ d\ \%\ \mathit{delay}\ (d_2 - d))\ \% \\
\qquad\qquad\qquad\quad (\mathit{stretch}\ (\mathit{extTailT}\ t\ d_1)\ d_1\ d)
\end{array}
$$

Lorsque $d_1 == d_2$, modulo normalisation, la fonction *stretch* est sans effet. Par contre, lorsque $d_2 < d_1$, le signal recomposé est plus court, son écoute sera donc apparemment plus rapide. Au contraire, lorsque $d_2 > d_1$, le signal recomposé est plus long, son écoute sera donc apparemment plus lente.

Le cas où $d_2 < d_1$ est illustré Figure 18.

Autrement dit, lorsque d est choisie singulièrement plus long que les périodes des notes apparaissant dans

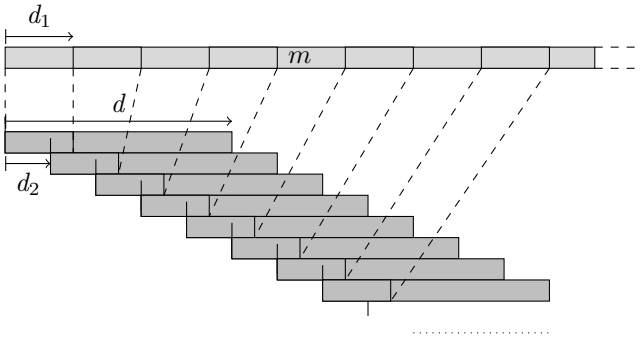

Figure 18. Contraction par décomposition et recomposition granulaire

les signaux manipulés, la hauteur de ces notes ne changera pas. La fonction *stretch* permet donc de ralentir ou accélérer un signal sonore sans (trop) en altérer l'écoute.

Phénomène bien connu de la synthèse granulaire, la texture sonore pourra cependant être largement altérée dès lors que la différence entre d_1 et d_2 est significative.

5. CONCLUSIONS

Dans les pages qui précèdent, nous avons décrit le modèle des signaux audio tuilés tel qu'il peut apparaître naturellement en enrichissant les signaux non tuilés de la compositions séquentielles avec les délais négatifs. C'est, en un sens, l'approche historiquement proposée par Desain et Honing [8] (voir aussi [14]). Nous montrons cependant comment cette approche peut être formalisée efficacement, en suivant les principes d'implémentation primitives des tuiles qui sont décrits dans [3].

Dans ce formalisme, la simplicité de codage du *stretch* par décomposition et re-composition à la volée, qui s'apparente aux techniques de synthèses granulaires, est plutôt une (bonne) surprise. Une expérimentation plus systématique de l'utilisation du tuilage audio pour le traitement du signal doit cependant être conduite. Le développement d'un convertisseur audio/tuile, paramétré par le taux d'échantillonnage reçu ou visé, est en cours. Ce convertisseur permettra une réelle expérimentation de notre approche à travers la lecture, la combinaison, la transformation et la production de fichiers audio.

Le principe d'évaluation paresseuse de Haskell couplée à la normalisation à la volée évoquée ci-dessus pourrait par ailleurs ouvrir la porte à son extension vers un player temps-réel. Les tests de performances actuellement en cours sont prometteurs.

6. REFERENCES

[1] C. Agon, J. Bresson, and G.Assayag. *The OM composer's Book, Vol.1 & Vol.2.* Collection Musique/Sciences. Ircam/Delatour, 2006.

[2] A. Allombert, M. Desainte-Catherine, and M. Toro. Modeling temporal constrains for a system of interactive score. In G. Assayag and C. Truchet, editors, *Constraint Programming in Music*, chapter 1, pages 1–23. Wiley, 2011.

[3] T. Bazin and D. Janin. Flux média tuilés polymorphes : une sémantique opérationnelle en Haskell. In *Journées Francophones des Langages Applicatifs (JFLA)*, 2015.

[4] F. Berthaut, D. Janin, and M. Desainte-Catherine. libTuile : un moteur d'exécution multi-échelle de processus musicaux hiérarchisés. In *Actes des Journées d'informatique Musicale (JIM)*, 05 2013.

[5] F. Berthaut, D. Janin, and B. Martin. Advanced synchronization of audio or symbolic musical patterns : an algebraic approach. *International Journal of Semantic Computing*, 6(4) :409–427, 12 2012.

[6] J. Bresson, C. Agon, and G. Assayag. Visual Lisp / CLOS programming in OpenMusic. *Higher-Order and Symbolic Computation*, 22(1), 3 2009.

[7] R.B. Dannenberg, C.L. Fraley, and P. Velikonja. A functional language for sound synthesis with behavioral abstraction and lazy evaluation. In Denis Baggi, editor, *Computer Generated Music*. IEEE Computer Society Press, 1992.

[8] P. Desain and H. Honing. LOCO : a composition microworld in Logo. *Computer Music Journal*, 12(3) :30–42, 1988.

[9] P. Hudak. A sound and complete axiomatization of polymorphic temporal media. Technical Report RR-1259, Department of Computer Science, Yale University, 2008.

[10] P. Hudak. *The Haskell School of Music : From signals to Symphonies*. Yale University, Department of Computer Science, 2013.

[11] P. Hudak, J. Hugues, S. Peyton Jones, and P. Wadler. A history of Haskell : Being lazy with class. In *Third ACM SIGPLAN History of Programming Languages (HOPL)*. ACM Press, 2007.

[12] P. Hudak and D. Janin. Programmer avec des tuiles musicales : le T-calcul en Euterpea. In *Actes des Journées d'informatique Musicale (JIM)*, 2014.

[13] P. Hudak and D. Janin. Tiled polymorphic temporal media. In *Work. on Functional Art, Music, Modeling and Design (FARM)*, pages 49–60. ACM Press, 2014.

[14] P. Hudak and D. Janin. From out-of-time design to in-time production of temporal media. Research report, LaBRI, Université de Bordeaux, 02 2015.

[15] D. Janin. Vers une modélisation combinatoire des structures rythmiques simples de la musique. *Revue Francophone d'Informatique Musicale (RFIM)*, 2, 09 2012.

[16] D. Janin, F. Berthaut, M. DeSainte-Catherine, Y. Orlarey, and S. Salvati. The T-calculus : towards a structured programming of (musical) time and space. In *Work. on Functional Art, Music, Modeling and Design (FARM)*, pages 23–34. ACM Press, 2013.

[17] F. Lerdahl and R. Jackendoff. *A generative theory of tonal music*. series on cognitive theory and mental representation. MIT Press, 1983.

[18] S. Letz, Y. Orlarey, and D. Fober. Real-time composition in Elody. In *Int. Computer Music Conference (ICMC)*, pages 336–339. ICMA, 2000.

[19] S. Letz et al. The LibAudioStream library, 2012. `http://libaudiostream.sourceforge.net/`.

[20] Y. Orlarey, D. Fober, and S. Letz. Faust : an efficient functional approach to DSP programming. In *New Computationals Paradigms for Computer Music*. Editions Delatour France, 2009.

[21] Y. Orlarey, D. Fober, and S. Letz. Faust : an efficient functional approach to DSP programming. In *New Computationals Paradigms for Computer Music*. Editions Delatour France, 2009.

[22] C. Roads. *L'audionumérique*. Dunod, 1998.

[23] M. Toro, M. Desainte-Catherine, and J. Castet. An extension of interactive scores for multimedia scenarios with temporal relations for micro and macro controls. In *Sound and Music Comp. (SMC)*, 2012.